MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 1366 June 1992

Why Do We See Three-Dimensional Objects?

Thomas Marill

Abstract

When we look at certain line-drawings, we see three-dimensional objects. The question is why. Why not just see two-dimensional images? Our theory is that we see the objects rather than the images because the objects are *simpler* than the images.

We define the complexity of an object as the number of bits in a pose-independent, binary representation of that object. We examine a number of examples and find that in each case the seen object is indeed simpler than the given image.

This leads us to our second question. Given that we are going to see a three-dimensional object when we look at a line-drawing, *which* three-dimensional object will we see? Our theory is that the vision system will pick the simplest object from among the infinite set of possibilities. We examine a number of examples and find that in each case the data is consistent with the theory.

This work is based on the pioneering ideas of Solomonoff and Kolmogorov, and on the more recent "minimum description length" concepts of Rissanen.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

1. Introduction.

When we look at certain line-drawings, we see three-dimensional objects. The question is why. Why see in three dimensions? Why not just see the two-dimensional images?

When we look at the two-dimensional line-drawing shown in figure 1, for example, we see a three-dimensional wire-frame cube. But there is no necessity about this. The image could just as well have been interpreted as a two-dimensional figure lying in the image plane. Why the three-dimensional perception?

Our theory is that we see the three-dimensional object rather than the two-dimensional image because the object is *simpler*. And, given the choice, the simpler solution will be preferred by the human vision system. The theory will be made clearer in what follows, and a number of examples will be considered in detail. We will see that our theory is supported by the data.

What do we mean by simplicity? As will be explained, we define the complexity of an object as the number of bits in a binary representation of that object. However, since we want the complexity of an object to remain constant when the object moves, we want our binary representation to be "pose-independent"; that is, we want the representation to be the same, regardless of the translation or rotation of the object. Thus, we need to construct a binary, pose-independent representation.

Why might the vision system prefer a simple (i.e., short) representation over a complex (long) one? There could be many reasons, but the most obvious one is that simple representations require less storage space. Since the cognitive system needs to remember many of its visual inputs, it seems reasonable to assume that storage economy is a primary consideration; this in turn dictates the use of simple representations.

Thus our theory suggests an answer to the question of why we see a three-dimensional object rather than a two-dimensional image when we look at figure 1. However, there remains a second, equally important question. Given that we are going to see a three-dimensional object, why a *cube*? After all, there are infinitely many three-dimensional wire-frame objects that project to figure 1. Why pick a cube?

As regards this second question, our theory is that, among all the objects that project to figure 1, the cube is the simplest. We will see that here too the data is consistent with the theory.

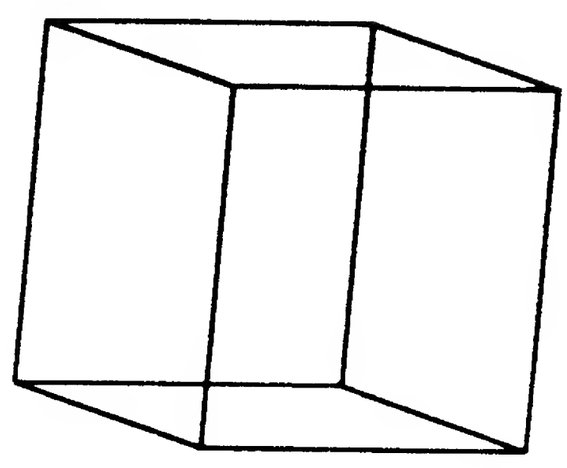

Fig. 1. A simple line-drawing.

In Section 2 we define the complexity of wire-frame objects and describe our multi-level representation scheme. In Section 3 we present our experimental results: we examine the complexity of thirteen specific line-drawings and of the objects we perceive when we look at these drawings. In Section 4 we discuss the historical antecedents of our ideas, and we look briefly at related work in computer vision. The Appendices give the numerical specification of the objects used in the experiments and the discussion.

Result-oriented readers may wish to avoid the technical details and turn directly to Section 3.

2. Descriptive Complexity and Levels of Representation.

We define the descriptive complexity (complexity, for short) of a wire-frame object as the number of bits in a pose-independent, binary representation of that object. If we think of this representation as a "code" for the object, the complexity is then simply the code-length.

We will actually deal with three levels of representation, of which the code is the innermost level. We start our discussion with what we call the "conventional" representation. At this level we are dealing with a pose-dependent, alphanumeric representation. From there we go to the second, or intermediate, level, which is a pose-independent, alphanumeric representation. Finally, we go to the binary, pose-independent representation, which allows us directly to measure descriptive complexity.

Programs for generating these representations have been implemented. In this software system representations at the first level are referred to as "Objects";

representations at the second, intermediate, level are referred to as "Forms"; and representations at the third, binary, level as "Codes".

These three levels are discussed in sections 2.1, 2.2, and 2.3. In section 2.4 we discuss the programs that translate among the three levels.

2.1 Objects: The conventional representation.

We first introduce what we call the conventional representation of objects and images. For example, the following expression represents a wire-frame cube of a particular size and in a particular location:

```
CUBE-OBJECT[1]
(OBJECT
:POINTS ((-2.25 -1.27 -0.79) (0.45 -1.27 -2.34) (0.72 1.81 -1.88) (-1.98 1.81
-0.32) (-0.72 -1.81 1.88) (1.98 -1.81 0.32) (2.25 1.27 0.79) (-0.45 1.27 2.34))
:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))
```

The list following ":POINTS" represents a list of three-dimensional points; these are the eight vertices of CUBE-OBJECT, expressed with respect to a fixed coordinate system. The list following ":LINES" represents a list of the twelve edges in the cube; each pair of integers on this list represents the indices of the points connected by a line; thus, the first pair, (0 1), means that the zeroth point, (-2.25 -1.27 -0.79), is connected by a straight-line segment to the first point, (0.45 -1.27 -2.34).

This representation of objects is intuitive and easy to work with. It has the drawback, however, that it is sensitive to the "pose" of the object; i.e., it varies as the object is translated or rotated with respect to the coordinate system. For example, if the above object is rotated 45 degrees around a vertical axis through its center, its representation becomes:

```
CUBE-OBJECT[2]
(OBJECT
```

:POINTS ((-2.15 -1.27 1.04) (-1.34 -1.27 -1.97)(-0.82 1.81 -1.83) (-1.63 1.81 1.18) (0.82 -1.81 1.83) (1.63 -1.81 -1.18) (2.15 1.27 -1.04) (1.34 1.27 1.97))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

We represent line-drawings images the same way we represent objects, except that in the case of images all z-coordinates are equal to zero. (That is, we consider images to be objects all of whose points lie in the x-y plane (the "image plane"). Thus, we treat images as a subset of objects.)

For example, the line-drawing in figure 1, is represented as follows. (This image is in fact the orthographic projection into the image-plane of CUBE-OBJECT[1].)

CUBE-IMAGE[1]

(OBJECT

:POINTS ((-2.25 -1.27 0.0) (0.45 -1.27 0.0) (0.72 1.81 0.0) (-1.98 1.81 0.0) (-0.72 -1.81 0.0) (1.98 -1.81 0.0) (2.25 1.27 0.0) (-0.45 1.27 0.0))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

If this image is translated or rotated in the image plane, its representation changes. For example if we rotate CUBE-IMAGE[1] by 45 degrees around its center, we get:

CUBE-IMAGE[2]

(OBJECT

:POINTS ((-0.7 -2.49 0.0) (1.21 -0.58 0.0) (-0.77 1.79 0.0) (-2.68 -0.12 0.0) (0.77 -1.79 0.0) (2.68 0.12 0.0) (0.7 2.49 0.0) (-1.21 0.58 0.0))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

2.2 Forms: The intermediate representation.

The intermediate representation, the "form," is based on the lengths of the lines

in a fully-connected (or "fully-triangulated") object, that is, an object in which all pairs of points are connected by lines.

For example, the form corresponding to CUBE-OBJECT[1], above, is as follows:

CUBE-FORM[1]

(FORM

:NUMBER-OF-POINTS 8

:VISIBLE-LINES ((0 1) (0 3) (0 4) (1 2) (1 5) (2 3) (2 6) (3 7) (4 5) (4 7) (5 6) (6 7))

:LINES-AND-LENGTHS (((0 1) 3.12) ((0 2) 4.41) ((0 3) 3.12) ((0 4) 3.12) ((0 5) 4.41) ((0 6) 5.4) ((0 7) 4.41) ((1 2) 3.12) ((1 3) 4.41) ((1 4) 4.41) ((1 5) 3.12) ((1 6) 4.41) ((1 7) 5.4) ((2 3) 3.12) ((2 4) 5.4) ((2 5) 4.41) ((2 6) 3.12) ((2 7) 4.41) ((3 4) 4.41) ((3 5) 5.4) ((3 6) 4.41) ((3 7) 3.12) ((4 5) 3.12) ((4 6) 4.41) ((4 7) 3.12) ((5 6) 3.12) ((5 7) 4.41) ((6 7) 3.12)))

This tells us (a) the number of points in a cube (eight); (b) the visible-lines, which are identical to the lines in the conventional representation (though not necessarily in the same order); and the lengths of *all* lines, i.e., the distances between all pairs of points (twenty-eight lengths). For example, the second element in the lines-and-lengths list is ((0 2) 4.41). This means that the distance from the zeroth to the second point is 4.41 units. This particular line is not among the lines of the original cube-object; i.e., it is not a "visible line."

This representation is independent of the pose of the object. Thus the form corresponding to CUBE-OBJECT[2], above, will be identical to CUBE-FORM[1].

The programs for generating forms from objects and objects from forms are discussed below in Section 2.4. Since forms are pose-independent, it is clear that if we go from object to form and then back to object, we will not, in general, get back the object in the original pose (see Section 2.4.4).

2.3 Codes: The binary representation.

The third level of representation, the "code," is the binary encoding of the form. The length of this code for a given object is our measure of descriptive complexity of the object.

For example, the code corresponding to CUBE-FORM[1] (and therefore to CUBE-OBJECT[1]) is:

CUBE-CODE[1]

(1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1 0 0 1 1 0 1 0 1 0 0 0 1 1 1 0 0 0 0 1 1 1 0
1 1 0 1 1 1 1 1 0 0 1 0 1 0 0 1 0 0 0 1 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 1 1 0 1
0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 1 1 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0
1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0
1 0 0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0 1
1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0 0 0 1 0 1 1 0 0)

Since the codes are independent of the pose of the original object, the code for CUBE-OBJECT[2] is identical to the above. (This identity, of course, is a consequence of the fact that the intermediate representations, on which the codes are based, are pose-independent.)

The programs for generating codes from forms and forms from codes are discussed in the following section.

2.4 Programs for generating representations.

The following programs were implemented for moving among the levels of representation: code-from-object (form-from-object followed by code-from-form); and object-from-code (form-from-code followed by object-from-form).

2.4.1 The program form-from-object.

The process for generating forms from objects is entirely straightforward. Basically, the process consists of first generating the "fully-connected" object, (expressed in the object-representation), then measuring the lengths of all the lines, and finally generating the lines-and-lengths list. Each element of this list consists, first, of a pair of integers, where each integer represents the index of an object-point (indices start with zero), and, second, the distance from the point designated by the first index to the point designated by the second index.

It is also necessary to record the number of points in the object and the "visible lines", i.e., the indices of the points in the original object that are connected by line-segments.

It is clear that the form type of representation is ambiguous with respect to mirror images. It is therefore possible, by going from an object to the form and then back to the object, that the mirror image object will be recovered. In our present study, this is not a problem since we are interested in the code length; and the code lengths of an object and its mirror image are equal.

2.4.2 The program code-from-form.

The process of generating codes from forms is somewhat more involved.

As we have seen, forms contain floating point numbers. There are many possible ways to encode such numbers. The approach taken here (following Rissanen [5]) is to specify a level of precision (number of decimal places), and to transform the floating-point numbers into integers by first multiplying by ten raised to the number of decimal places and then rounding to the nearest integer. The integers are then encoded. Thus, our codes are conditioned on the number of decimal places selected. In this study, the number is arbitrarily fixed at four.

However, since integers consist of a variable number of digits, even the encoding of integers is not entirely straightforward. Still following Rissanen, we use a "universal code" devised by Elias [1]. This code allows us to concatenate the codes for the integers in a list (without restriction on the size of the integers) and to decode the resulting binary string into the original list of integers.

In order to obtain economy in coding, since many of the integers tend to repeat for a given object, the program constructs a dictionary of all of the integers to be encoded. Then, instead of encoding a particular integer, the program encodes the index of the entry for that integer in the dictionary (this index, of course, is itself an integer). The program also encodes the dictionary itself.

The example of CUBE-CODE[1] seen above can be understood in terms of the following partitioning.

CUBE-CODE[1]

(— 1 1 1 0 0 0 0 — 1 1 1 1 0 0 0 — 1 0 1 0 1 0 — 0 1 1 0 1 0 1 0 0 0 1 1 1 0 0
0 0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 0 0 1 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 0 0 1
0 1 1 1 0 1 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 1 1 0 0 0 — 0 1 0 0
0 0 1 0 0 1 1 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 0
0 0 0 1 0 0 0 0 1 0 0 0 — 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0
1 0 1 1 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0 0 0 1 0 1 1 0 0 —)

The first partition (1 1 1 0 0 0 0) is the code for the number of points (eight). The second (1 1 1 1 0 0 0) is the code for the number of visible-lines (twelve). The third partition (1 0 1 0 1 0) is the code for one plus the number of decimal places (five). The fourth partition is the encoding of the visible-lines list. The fifth partition is the encoding of the line-lengths. The final partition encodes the dictionary (which in this example is (3.1199 4.4122 5.4038) after rounding to four decimal places but prior to conversion to integers). In the present example the total code length, hence the complexity of CUBE-OBJECT[1], is 245.

2.4.3 The program form-from-code.

The program form-from-code is quite straightforward and requires very little discussion. The program reconstitutes the form from the code given as argument. Since the code style of representation contains the codes of the original floating-point numbers only up to a given level of precision, the decoding process reconstitutes the form only up to that level of precision. Thus, if the number of decimal places specified to the program code-from-form is four, the decoded form will contain the original floating-point numbers rounded to four decimal places.

2.4.4 The program object-from-form.

The program for reconstructing an object from a form is somewhat complicated, and we will describe it here only in outline.

The problem that the program is required to solve can be thought of as follows. We are given a set of sticks of various lengths; each stick has a small integer written at each end (these correspond to points); the program must assemble all the sticks in such a way that wherever sticks touch the touching ends have the same integers written on them.

The object is constructed iteratively, one point at a time. To add a new point to a partially constructed object having four or more points, we pick three "sticks", each of which has one end at the already-constructed object and the other end at the desired new point. Thus, we have a tetrahedron of which we know the location of three points and the line-lengths from these three points to the fourth point. We can solve for the location of the fourth point, but there are two solutions. However, in general, only one of these is consistent with other line-lengths from the fourth point to other points in the already-constructed part of the object. By looking at these other lengths, we determine which of the solutions is the correct one.

It is immediately clear, since the form is pose-independent, that the program object-from-form cannot return the object in the original pose. What the program does is to return an object in a "standard pose". The first point (the one corresponding to the index 0) is placed at the origin. The second point (corresponding to the index 1) is placed on the positive z-axis. The third point is placed on the positive half of the z-y plane. The fourth point has a negative x-coordinate. The fact that object-from-form does not return the object in its original pose is not a problem for us. We are basically interested only in the length of the code representation. The only reason we need the program object-from-form at all is to prove to ourselves that the codes were correctly generated.

3. Experiments dealing with the complexity of images and perceived objects.

3.1 Introduction to experiments.

In this section we discuss two experiments dealing with line-drawing images and the three-dimensional objects perceived when we look at these images.

In these experiments we use thirteen object-models. These are illustrated in figures 2, 3, and 4, and precise numerical specifications of these models in a particular pose are given in Appendix I. The line-drawings used are illustrated in the left-hand column of figures 2, 3 and 4. They are obtained by orthographic projection from the models in Appendix I in the particular pose given. (Thus, to obtain the numerical specification of these line-drawings, set the z-coordinates of the models in Appendix I to zero).

We are concerned with the three-dimensional objects that are perceived when we look at these line-drawings. What are these? We have reason to think that the perceived objects, in these thirteen cases, are identical or approximately identical to our thirteen models.

It must be emphasized that this is by no means always the case: it is easy to design examples in which the object seen when we look at a line-drawing is not at all the same as the model that generated the drawing. For example, consider the model COMPARISON-OBJECT-A, given in Appendix II and illustrated in figure 5. In this case, the object we perceive when we look at the leftmost image in figure 5 is strikingly *dissimilar* to the model that generated the image. (Further examples are given by Marill [4].)

However, experimentation demonstrates that if the perceived object is not identical to the model, then, when the model is rotated, the object will be perceived as deforming. This provides us with a test: if a rotating model is perceived as solid (non-deforming), we can be reasonably sure that the object perceived when we look at one of its views is the same as the model. In fact, the thirteen models in our experiments pass this test.

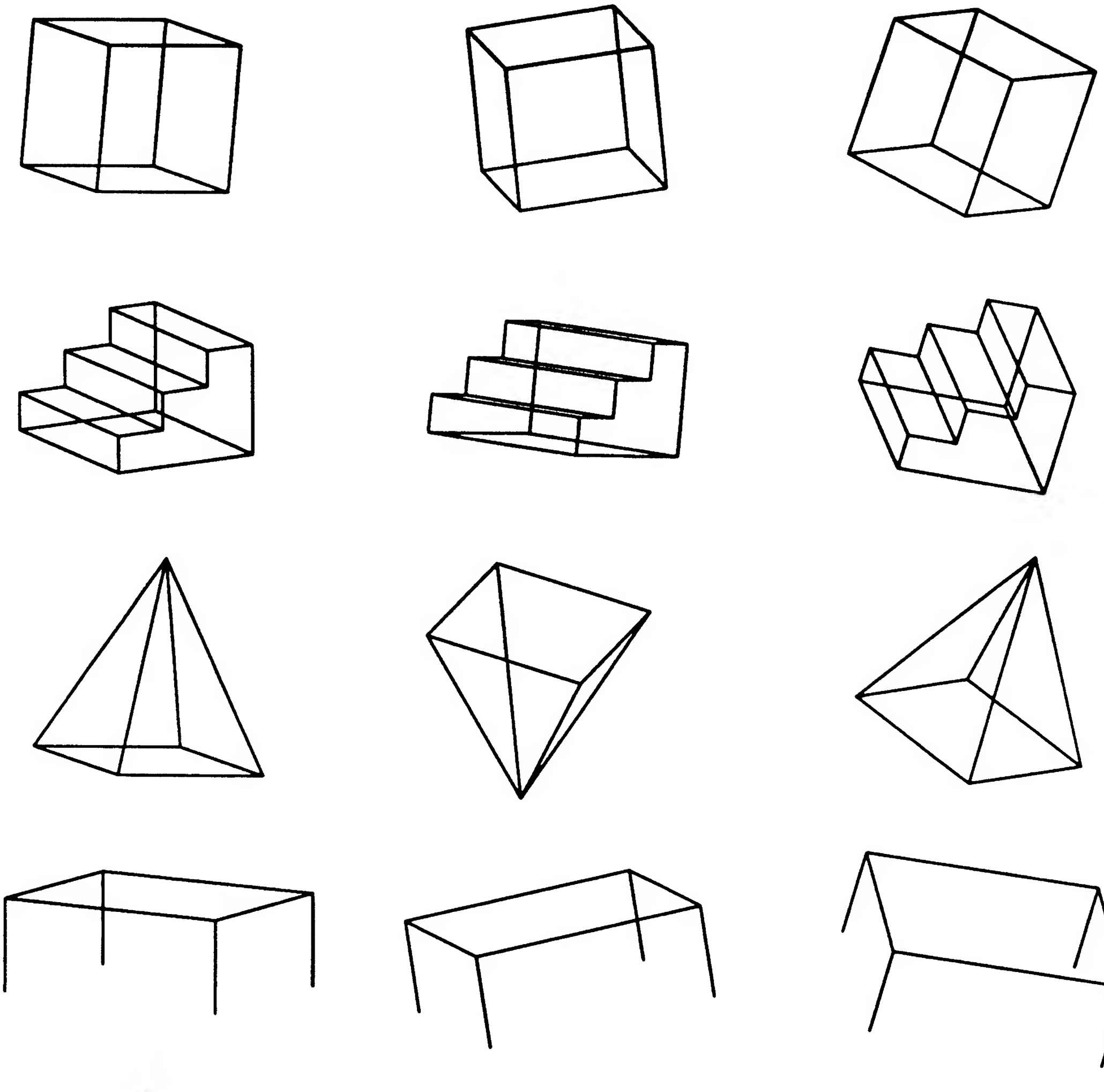

Fig. 2. Three views of CUBE, STAIRCASE, PYRAMID, TABLE and SLANTED-SQUARE.

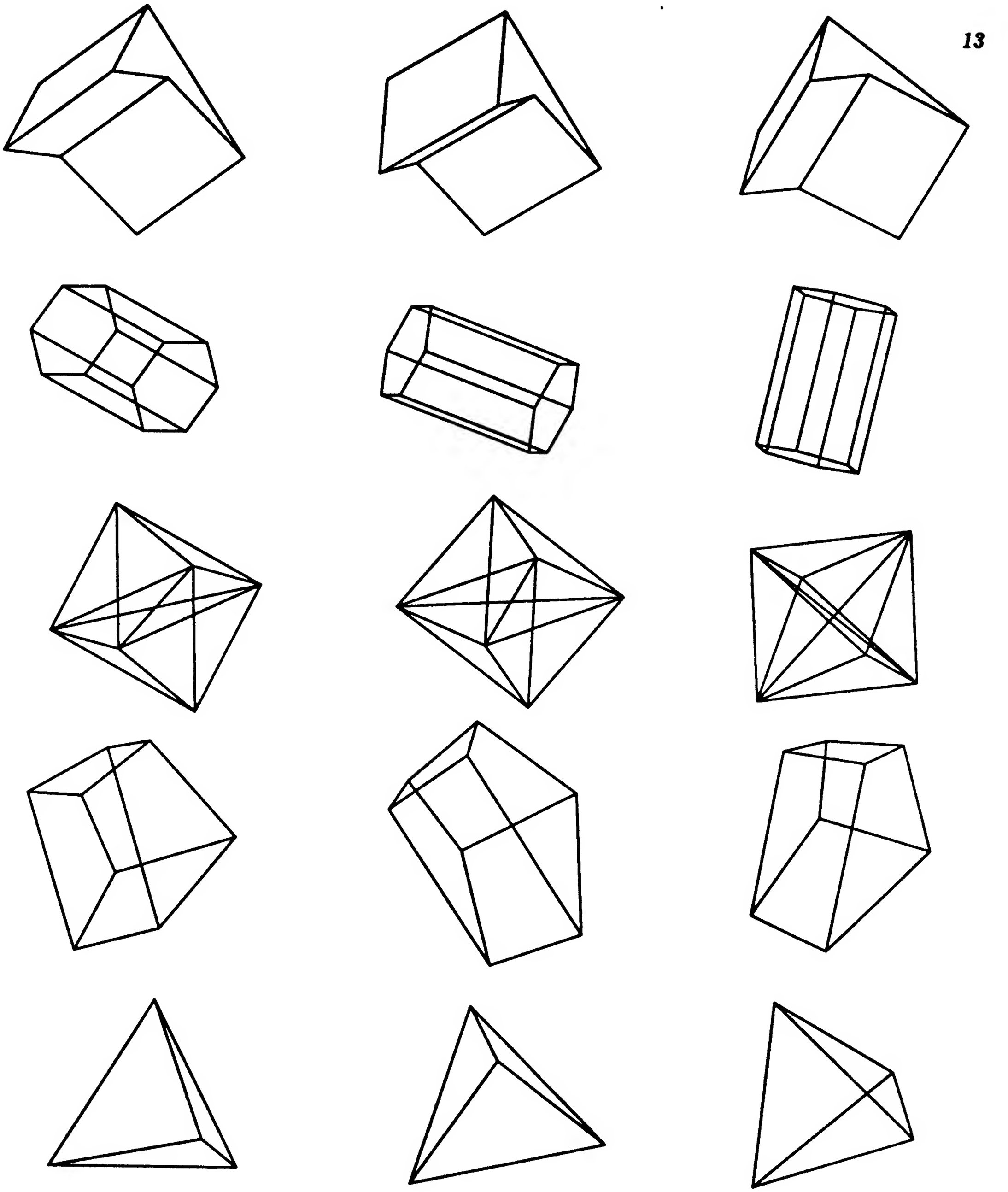

Fig. 3. Three views of CONCAVE-WEDGE, HEX-PRISM, SPACE-STATION ASYMMETRIC-OBJECT and REGULAR-TET

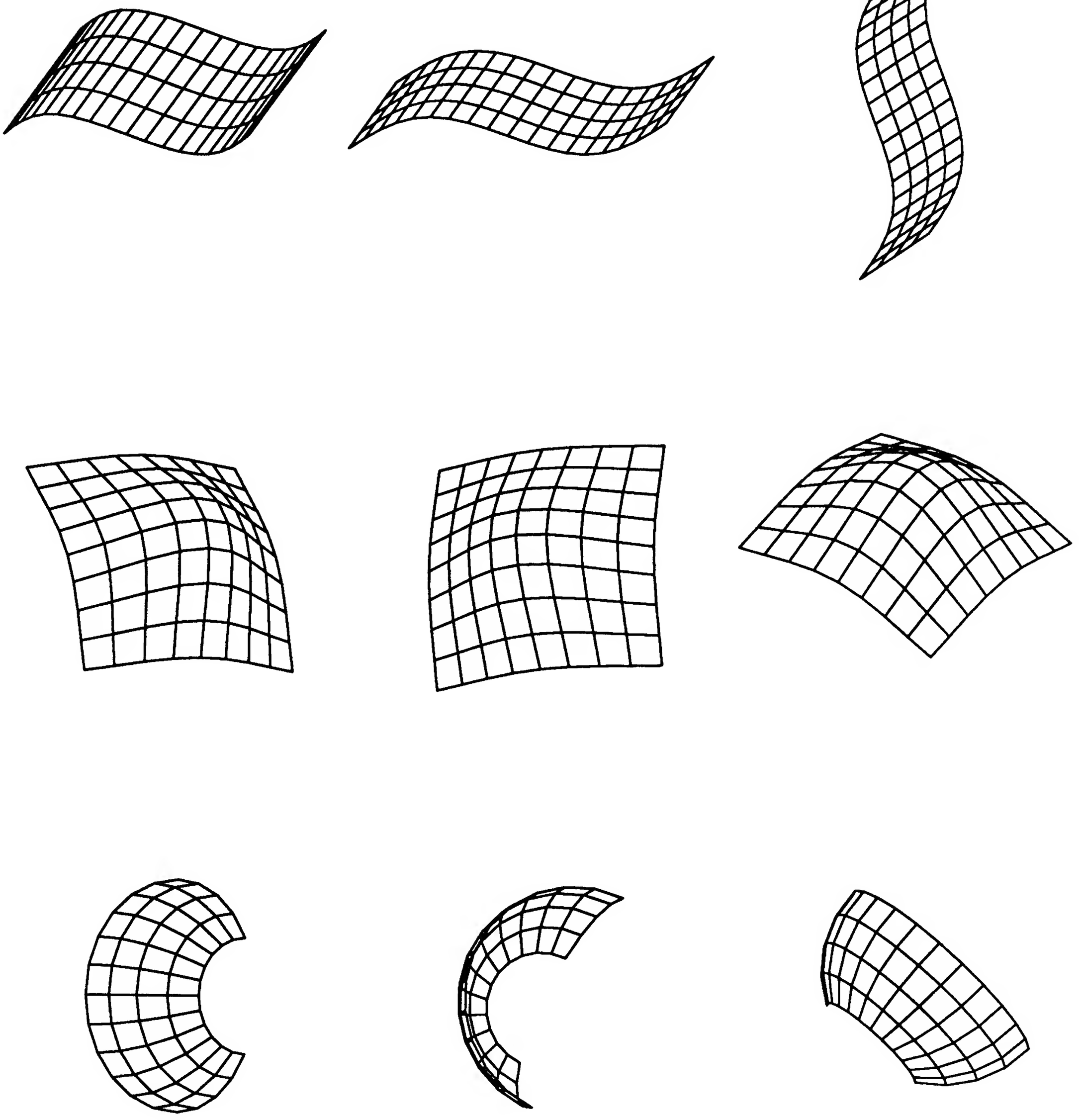

Fig. 4. Three views of SINE-SURFACE, GAUSSIAN-SURFACE, and SPHERE-PATCH

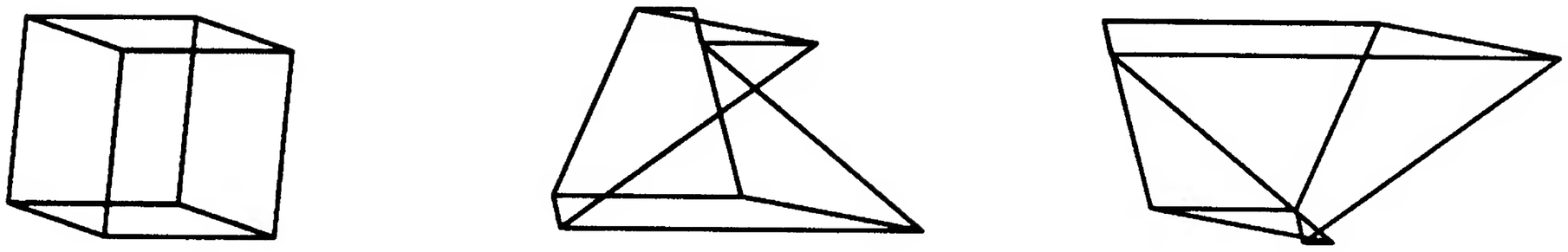

Fig. 5. Three views of COMPARISON-OBJECT-A. The representation is given in Appendix II. In the second and third views, the object has been rotated plus and minus ten degrees. For a discussion of the complexity of this object, see Appendix II.

3.2 The complexity of images and perceived objects.

Based on the preceding, we assume that the thirteen three-dimensional objects listed in Appendix I are the objects that are perceived when we look at the left-hand images in figures 2, 3, and 4. Table 1 gives the descriptive complexity of the thirteen images and of the corresponding perceived objects.

All images and objects were first expressed in the conventional representation (Appendix I). They were then translated into the binary codes discussed above, using four decimal places of accuracy. The number of bits of the binary codes, as explained, were used as the measure of complexity.

It will be seen that in every instance the complexity of the image is greater than the complexity of the object. That is, the objects are simpler, in conformity with our theory. This may serve to explain why we see the three-dimensional object rather than the two-dimensional image.

For a discussion of the complexity of COMPARISON-OBJECT-A (figure 5), see Appendix II.

Object	Object Complexity	Image Complexity
CUBE	245	560
STAIRCASE	1787	2497
PYRAMID	153	288
TABLE	362	518
SLANTED-SQUARE	104	157
CONCAVE-WEDGE	433	727
HEX-PRISM	647	1324
SPACE-STATION	166	376
ASYMMETRIC-OBJECT	924	1027
REGULAR-TET	72	202
SINE-SURFACE	68326	87108
GAUSSIAN-SURFACE	57918	119550
SPHERE-PATCH	28454	57463

Table 1. The complexity of objects and their images.

3.3 The complexity of distorted objects whose images are held constant.

The preceding experiment suggests an answer to the question of why we see three-dimensional objects rather than two-dimensional images. But it says nothing about *which* three-dimensional objects we see.

In the experiment discussed in the present section, we use a *distortion factor* to distort objects while holding their images constant. It is clear that if we start with an object in a fixed pose (an object in our first representation), we can change the z-coordinates of any of the points without changing the orthographic projection of the object. The distortion factor is a single number which multiplies all the z-coordinates in an object; as a result of the multiplication, the object distorts while its orthographic projection remains unchanged.

For each of the objects used in the preceding experiment (by assumption, these are the objects perceived when we look at the thirteen images used in the preceding experiments), we applied six distortion factors: 0, 0.25, 0.75, 1, 1.5, and 5. A distortion factor of zero sets all z-coordinates equal to zero, and therefore makes the object identical to the image. A distortion factor of one leaves all z-coordinates unchanged, and therefore leaves the object undistorted. Each of the resulting seventy-eight objects were converted to binary code, and the lengths of the codes are presented in Table 2.

It will be seen that in all thirteen cases the minimum of each row occurs at a distortion factor of one. This means that the simplest object is the undistorted object, i.e., by our assumption, the object that is perceived when we look at the image. This suggests that the vision system, in accordance with our theory, selects the simplest object, that is, the one with smallest complexity, from among the possible objects that project to the image.

	Distortion factor					
	0	0.25	0.75	1	1.5	5
CUBE	560	553	545	245	557	627
STAIRCASE	2497	2471	2419	1787	2460	2664
PYRAMID	288	288	288	153	298	335
TABLE	518	519	519	362	534	592
SLANTED-SQUARE	157	158	162	104	172	179
CONCAVE-WEDGE	727	725	725	433	757	827
HEX-PRISM	1324	1320	1303	647	1338	1446
SPACE-STATION	376	376	375	166	381	423
ASYMMETRIC-OBJECT	1027	1038	1060	924	1084	1149
REGULAR-TET	202	202	205	72	206	233
SINE-SURFACE	87108	87076	86904	68326	87828	91908
GAUSSIAN-SURFACE	119550	119288	120566	57918	122972	130572
SPHERE-PATCH	57463	57400	57543	28454	57991	60235

Table 2. The complexity of distorted objects whose images are held constant.

4. Historical antecedents and related work.

4.1 Historical antecedents.

The present work is based on the pioneering ideas of Solomonoff [6] and Kolmogorov [2], and on the more recent "minimum description length" concepts of Rissanen [5].

Solomonoff was concerned with the problem of induction. Given a long sequence of symbols, can we predict the next symbol? Solomonoff defined the concept of the *description* of a sequence S to be a binary string which, when input to a certain machine, will cause that machine to generate S as output. He was able to show that the description-length (number of bits in the description of S) is related to the probability of S, and therefore to the probability of any next symbol.

Kolmogorov's interest lay in the foundations of probability and information theory. He was able to show that the length of the shortest binary "program" that generates as output an arbitrary sequence S is related to the probability and the information content of S. Unfortunately, he also showed that the problem of finding such shortest programs is non-computable.

Thus, in both Solomonoff and Kolmogorov we find the concept of description length at the foundation of the theory. Some twenty years later, the idea was incorporated into Rissanen's principle of minimum description length (MDL), with application to statistical parameter estimation.

In Rissanen's approach, one assumes that the observable data comes from one member of a class of known models. (It is this restrictive assumption that allows the approach to generate computable results.) To estimate the parameters of the model that generated the data, we first formulate an expression for the model and the deviations of the data from the model. We then minimize the binary representation of this expression over the space of parameters; this yields the parameter estimates.

4.2 Discussion.

All of these techniques (as well as ours in the present paper) suffer from one short-

coming: they are contingent on the choice of coding technique. Thus Rissanen uses (as do we) a particular code, due to Elias [1], for representing the integers. A different coding scheme would give different results. It is necessary, therefore, for the reader to think of the definition of the underlying principle as incorporating the particular coding scheme used. (In our case, the reader should think of our definition of complexity as incorporating our particular coding scheme.)

It is an essential requirement in all of these approaches that the description of the given data must be a true code from which the data can be recovered. That is, the number of bits must be the length of an invertible representation. We adhere to this requirement in the present work (even though, for our very restricted purpose, it might not be absolutely essential).

4.3 Related vision research.

In a recent paper, Leclerc [3] demonstrated the power of some of these ideas in the realm of computer vision. When we look at a grey-scale image, such as a black-and-white photograph of a natural scene, an important aspect of our perception consist of boundaries between regions (discontinuities between areas in which certain measurements change continuously). Leclerc was able to show that these discontinuities could be found by proper encoding of the data and the model, and by then minimizing the code-length. His results appear to agree with the perceptions of the human vision system.

In a more recent paper, Marill [4] dealt with the three-dimensional perception of line-drawings. The problem was to devise a program which, given the representation of a line-drawing, would return the representation of the same three-dimensional object that a human observer sees when looking at that line-drawing.

To solve this problem, Marill proposed the idea that, in generating percepts of three-dimensional objects, the human vision system minimizes a certain aspect of the object. He demonstrated that the principle of minimum standard-deviation of angles (MSDA) was able to generate good results over a variety of examples. In accordance with the MSDA principle, one selects, from among the infinite number of three-dimensional objects that project to the given image, that particular object for which the standard-deviation of the angles is minimized.

It is entirely possible that the MSDA principle is a special case of the process

presented in the present paper. However, that remains to be proved.

Finally, it should be pointed out that while the present paper restricts itself to questions of two-dimensional images and three-dimensional objects, there are also interesting questions to be asked about dimensions higher than three. Why do we see three-dimensional objects rather than four-dimensional objects or objects of higher dimensions? Are three-dimensional objects simpler than the higher-dimensional objects of which they are the projections? For now, such questions must remain open.

Acknowledgments.

I would like to thank Boris Katz, who struggled against long odds to make this a better paper, as well as Rodney Brooks and David McAllester, who both provided illuminating comments.

Bibliography.

[1] Elias, P. (1975). Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory. IT-21(2), 194-203.

[2] Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. Problemy Peredachi Informatsii, 1(1), 3-11.

[3] Leclerc, Y. G. (1989). Constructing simple stable descriptions for image partitioning. International Journal of Computer Vision, 3(1), 73-102.

[4] Marill, T. (1991). Emulating the human interpretation of line-drawings as three-dimensional objects. International Journal of Computer Vision, 6(2), 147-161.

[5] Rissanen, J. (1983). A universal prior for integers and estimation by minimum description length. The Annals of Statistics, 6(2), 416-431.

[6] Solomonoff, R. J. (1964). A formal theory of inductive inference. Information and Control, 7. Part I, 1-22; Part II, 224-254.

Appendix I.

The following are the representations of the thirteen three-dimensional wire-frame model-objects used in the experiments discussed above. To obtain the images used in the experiments, set the z-coordinates to zero.

CUBE

(OBJECT

:POINTS ((-2.25 -1.27 -0.79) (0.45 -1.27 -2.34) (0.72 1.81 -1.88) (-1.98 1.81 -0.32) (-0.72 -1.81 1.88) (1.98 -1.81 0.32) (2.25 1.27 0.79) (-0.45 1.27 2.34))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

STAIRCASE

(OBJECT

:POINTS ((-1.74 -1.24 0.74) (0.86 -0.85 -0.71) (0.86 1.33 -0.13) (0.0 1.2 0.36) (0.0 0.47 0.16) (-0.87 0.34 0.65) (-0.87 -0.38 0.45) (-1.74 -0.51 0.93) (0.14 -2.08 3.88) (2.74 -1.69 2.43) (2.74 0.49 3.01) (1.87 0.36 3.49) (1.87 -0.37 3.3) (1.0 -0.5 3.78) (1.0 -1.22 3.59) (0.14 -1.35 4.07))

:LINES ((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 0) (8 9) (9 10) (10 11) (11 12) (12 13) (13 14) (14 15) (15 8) (0 8) (1 9) (2 10) (3 11) (4 12) (5 13) (6 14) (7 15)))

PYRAMID

(OBJECT

:POINTS ((-0.83 -1.33 3.56) (2.63 -1.33 1.56) (0.66 -0.64 -1.86) (-2.8 -0.64 0.14) (0.35 3.94 1.6))

:LINES ((0 1) (1 2) (2 3) (3 0) (0 4) (1 4) (2 4) (3 4)))

TABLE

(OBJECT

:POINTS ((-0.67 1.5 3.76) (3.67 1.07 1.3) (1.67 0.47 -2.11) (-2.67 0.9 0.35) (-0.67 -0.47 4.11) (3.67 -0.9 1.65) (1.67 -1.5 -1.76) (-2.67 -1.07 0.7))

:LINES ((0 1) (1 2) (2 3) (3 0) (0 4) (1 5) (2 6) (3 7)))

SLANTED-SQUARE

(OBJECT

:POINTS ((1.52 -3.08 -1.06) (1.52 2.12 -4.06) (0.48 5.08 1.06) (0.48 -0.12 4.06))

:LINES ((0 1) (1 2) (2 3) (3 0) (0 2) (1 3)))

CONCAVE-WEDGE

(OBJECT

:POINTS ((-0.49 -0.11 -0.75) (1.6 -2.32 -1.62) (-1.04 1.92 -2.64) (-2.02 0.07 2.01) (2.49 2.11 0.75) (4.57 -0.1 -0.12) (1.93 4.14 -1.14) (0.95 2.29 3.51))

:LINES ((0 1) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

HEX-PRISM

(OBJECT

:POINTS ((-0.33 1.33 0.33) (-0.14 0.51 -0.22) (-0.71 -0.3 -0.13) (-1.48 -0.3 0.51) (-1.67 0.51 1.06) (-1.1 1.33 0.97) (1.48 0.3 2.49) (1.67 -0.51 1.94) (1.1 -1.33 2.03) (0.33 -1.33 2.67) (0.14 -0.51 3.22) (0.71 0.3 3.13))

:LINES ((0 1) (1 2) (2 3) (3 4) (4 5) (0 5) (6 7) (7 8) (8 9) (9 10) (10 11) (6 11) (0 6) (1 7) (2 8) (3 9) (4 10) (5 11)))

SPACE-STATION

(OBJECT

:POINTS ((-2.51 -0.59 -1.15) (0.9 1.03 -2.48) (2.51 0.59 1.15) (-0.9 -1.03 2.48) (0.93 -2.57 -0.73) (-0.93 2.57 0.73))

:LINES ((1 2) (2 3) (0 1) (0 3) (4 0) (4 1) (4 2) (4 3) (5 0) (5 1) (5 2) (5 3) (0 2) (1 3)))

ASYMMETRIC-OBJECT

(OBJECT

:POINTS ((-2.39 -5.85 3.74) (-0.77 -2.67 -2.99) (-2.12 0.46 -3.91) (-4.25 0.56 0.09) (1.0 -4.93 5.96) (4.04 -1.18 -1.71) (0.6 2.71 -1.89) (-1.1 2.41 1.72))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (0 4) (1 5) (2 6) (3 7) (4 7)))

REGULAR-TET

(OBJECT

:POINTS ((-0.22 0.16 0.56) (3.54 0.16 -0.81) (1.86 3.57 0.44) (2.83 0.73 3.08))

:LINES ((0 1) (0 2) (0 3) (1 2) (1 3) (2 3)))

SINE-SURFACE

(OBJECT

:POINTS ((-3.35 -1.33 1.75) (-2.95 -0.77 2.15) (-2.55 -0.2 2.55) (-2.15 0.37 2.95) (-1.75 0.93 3.35) (-3.17 -1.19 1.36) (-2.77 -0.62 1.76) (-2.37 -0.05 2.16) (-1.97 0.51 2.56) (-1.57 1.08 2.96) (-2.97 -1.06 0.99) (-2.57 -0.49 1.39) (-2.17 0.07 1.79) (-1.77 0.64 2.19) (-1.37 1.2 2.59) (-2.76 -0.96 0.64) (-2.36 -0.4 1.04) (-1.96 0.17 1.44) (-1.56 0.73 1.84) (-1.16 1.3 2.24) (-2.51 -0.91 0.32) (-2.11 -0.35 0.72) (-1.71 0.22 1.12) (-1.31 0.78 1.52) (-0.91 1.35 1.92) (-2.23 -0.91 0.04) (-1.83 -0.35 0.44) (-1.43 0.22 0.84) (-1.03 0.78 1.24) (-0.63 1.35 1.64) (-1.91 -0.96 -0.21) (-1.51 -0.4 0.19) (-1.11 0.17 0.59) (-0.71 0.73 0.99) (-0.31 1.3 1.39) (-1.56 -1.06 -0.43) (-1.16 -0.49 -0.03) (-0.76 0.07 0.37) (-0.36 0.64 0.77) (0.04 1.2 1.17) (-1.19 -1.19 -0.62) (-0.79 -0.62 -0.22) (-0.39 -0.05 0.18) (0.01 0.51 0.58) (0.41 1.08 0.98) (-0.8 -1.33 -0.8) (-0.4 -0.77 -0.4) (0.0 -0.2 0.0) (0.4 0.37 0.4) (0.8 0.93 0.8) (-0.41 -1.48 -0.98) (-0.01 -0.91 -0.58) (0.39 -0.35

-0.18) (0.79 0.22 0.22) (1.19 0.79 0.62) (-0.04 -1.6 -1.17) (0.36 -1.04 -0.77) (0.76 -0.47 -0.37) (1.16 0.09 0.03) (1.56 0.66 0.43) (0.31 -1.7 -1.39) (0.71 -1.13 -0.99) (1.11 -0.57 -0.59) (1.51 0.0 -0.19) (1.91 0.56 0.21) (0.63 -1.75 -1.64) (1.03 -1.18 -1.24) (1.43 -0.62 -0.84) (1.83 -0.05 -0.44) (2.23 0.51 -0.04) (0.91 -1.75 -1.92) (1.31 -1.18 -1.52) (1.71 -0.62 -1.12) (2.11 -0.05 -0.72) (2.51 0.51 -0.32) (1.16 -1.7 -2.24) (1.56 -1.13 -1.84) (1.96 -0.57 -1.44) (2.36 0.0 -1.04) (2.76 0.56 -0.64) (1.37 -1.6 -2.59) (1.77 -1.04 -2.19) (2.17 -0.47 -1.79) (2.57 0.09 -1.39) (2.97 0.66 -0.99) (1.57 -1.48 -2.96) (1.97 -0.91 -2.56) (2.37 -0.35 -2.16) (2.77 0.22 -1.76) (3.17 0.79 -1.36) (1.75 -1.33 -3.35) (2.15 -0.77 -2.95) (2.55 -0.2 -2.55) (2.95 0.37 -2.15) (3.35 0.93 -1.75))

:LINES ((0 5) (5 10) (10 15) (15 20) (20 25) (25 30) (30 35) (35 40) (40 45) (45 50) (50 55) (55 60) (60 65) (65 70) (70 75) (75 80) (80 85) (85 90) (1 6) (6 11) (11 16) (16 21) (21 26) (26 31) (31 36) (36 41) (41 46) (46 51) (51 56) (56 61) (61 66) (66 71) (71 76) (76 81) (81 86) (86 91) (2 7) (7 12) (12 17) (17 22) (22 27) (27 32) (32 37) (37 42) (42 47) (47 52) (52 57) (57 62) (62 67) (67 72) (72 77) (77 82) (82 87) (87 92) (3 8) (8 13) (13 18) (18 23) (23 28) (28 33) (33 38) (38 43) (43 48) (48 53) (53 58) (58 63) (63 68) (68 73) (73 78) (78 83) (83 88) (88 93) (4 9) (9 14) (14 19) (19 24) (24 29) (29 34) (34 39) (39 44) (44 49) (49 54) (54 59) (59 64) (64 69) (69 74) (74 79) (79 84) (84 89) (89 94) (0 1) (1 2) (2 3) (3 4) (5 6) (6 7) (7 8) (8 9) (10 11) (11 12) (12 13) (13 14) (15 16) (16 17) (17 18) (18 19) (20 21) (21 22) (22 23) (23 24) (25 26) (26 27) (27 28) (28 29) (30 31) (31 32) (32 33) (33 34) (35 36) (36 37) (37 38) (38 39) (40 41) (41 42) (42 43) (43 44) (45 46) (46 47) (47 48) (48 49) (50 51) (51 52) (52 53) (53 54) (55 56) (56 57) (57 58) (58 59) (60 61) (61 62) (62 63) (63 64) (65 66) (66 67) (67 68) (68 69) (70 71) (71 72) (72 73) (73 74) (75 76) (76 77) (77 78) (78 79) (80 81) (81 82) (82 83) (83 84) (85 86) (86 87) (87 88) (88 89) (90 91) (91 92) (92 93) (93 94)))

GAUSSIAN-SURFACE

(OBJECT

:POINTS ((-3.08 -4.18 4.63) (-3.2 -3.16 4.42) (-3.32 -2.14 4.22) (-3.47 -1.16 3.96) (-3.68 -0.25 3.6) (-3.97 0.57 3.1) (-4.32 1.32 2.49) (-4.7 2.03 1.82) (-5.08 2.75 1.17) (-2.09 -4.03 4.35) (-2.14 -2.93 4.26) (-2.18 -1.83 4.19) (-2.27 -0.78 4.03) (-2.46 0.16 3.7) (-2.77 0.95 3.16) (-3.18 1.64 2.46) (-3.64 2.26 1.67) (-4.09 2.9 0.89) (-1.09 -3.87 4.08) (-1.07 -2.69 4.12) (-1.03 -1.5 4.18) (-1.07 -0.39 4.12) (-1.24 0.57 3.83) (-1.57 1.35 3.25) (-2.03 1.96 2.45) (-2.57 2.5 1.52) (-3.09 3.05 0.62) (-0.12 -3.76 3.76) (-0.04 -2.51 3.9) (0.05 -1.25 4.05) (0.06 -0.08 4.07) (-0.09 0.9 3.81) (-0.44 1.65 3.21) (-0.95 2.21 2.32) (-1.54 2.68 1.3) (-2.12 3.17 0.3) (0.79 -3.71 3.33) (0.89 -2.44 3.5)

(1.0 -1.16 3.69) (1.03 0.03 3.74) (0.88 1.02 3.49) (0.53 1.76 2.88) (0.0 2.31 1.96) (-0.61 2.75 0.9) (-1.21 3.22 -0.14) (1.61 -3.76 2.76) (1.69 -2.51 2.9) (1.78 -1.25 3.05) (1.79 -0.08 3.07) (1.64 0.9 2.81) (1.29 1.65 2.21) (0.78 2.21 1.32) (0.19 2.68 0.3) (-0.39 3.17 -0.7) (2.38 -3.87 2.08) (2.4 -2.69 2.12) (2.43 -1.5 2.18) (2.4 -0.39 2.12) (2.23 0.57 1.83) (1.9 1.35 1.25) (1.43 1.96 0.45) (0.9 2.5 -0.48) (0.38 3.05 -1.38) (3.11 -4.03 1.35) (3.06 -2.93 1.26) (3.01 -1.83 1.19) (2.92 -0.78 1.03) (2.73 0.16 0.7) (2.42 0.95 0.16) (2.01 1.64 -0.54) (1.56 2.26 -1.33) (1.11 2.9 -2.11) (3.85 -4.18 0.63) (3.73 -3.16 0.42) (3.61 -2.14 0.22) (3.46 -1.16 -0.04) (3.25 -0.25 -0.4) (2.96 0.57 -0.9) (2.61 1.32 -1.51) (2.23 2.03 -2.18) (1.85 2.75 -2.83))

:LINES ((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (9 10) (10 11) (11 12) (12 13) (13 14) (14 15) (15 16) (16 17) (18 19) (19 20) (20 21) (21 22) (22 23) (23 24) (24 25) (25 26) (27 28) (28 29) (29 30) (30 31) (31 32) (32 33) (33 34) (34 35) (36 37) (37 38) (38 39) (39 40) (40 41) (41 42) (42 43) (43 44) (45 46) (46 47) (47 48) (48 49) (49 50) (50 51) (51 52) (52 53) (54 55) (55 56) (56 57) (57 58) (58 59) (59 60) (60 61) (61 62) (63 64) (64 65) (65 66) (66 67) (67 68) (68 69) (69 70) (70 71) (72 73) (73 74) (74 75) (75 76) (76 77) (77 78) (78 79) (79 80) (0 9) (1 10) (2 11) (3 12) (4 13) (5 14) (6 15) (7 16) (8 17) (9 18) (10 19) (11 20) (12 21) (13 22) (14 23) (15 24) (16 25) (17 26) (18 27) (19 28) (20 29) (21 30) (22 31) (23 32) (24 33) (25 34) (26 35) (27 36) (28 37) (29 38) (30 39) (31 40) (32 41) (33 42) (34 43) (35 44) (36 45) (37 46) (38 47) (39 48) (40 49) (41 50) (42 51) (43 52) (44 53) (45 54) (46 55) (47 56) (48 57) (49 58) (50 59) (51 60) (52 61) (53 62) (54 63) (55 64) (56 65) (57 66) (58 67) (59 68) (60 69) (61 70) (62 71) (63 72) (64 73) (65 74) (66 75) (67 76) (68 77) (69 78) (70 79) (71 80)))

SPHERE-PATCH

(OBJECT

:POINTS ((0.25 3.89 -0.78) (-0.55 3.75 -0.11) (-1.28 3.35 0.51) (-1.92 2.72 1.04) (-2.41 1.89 1.45) (-2.71 0.92 1.71) (-2.82 -0.11 1.79) (-2.71 -1.15 1.71) (-2.41 -2.11 1.45) (-1.92 -2.94 1.04) (-1.28 -3.58 0.51) (-0.55 -3.98 -0.11) (0.25 -4.11 -0.78) (0.91 3.75 0.02) (0.15 3.62 0.66) (-0.57 3.23 1.26) (-1.18 2.62 1.77) (-1.65 1.82 2.17) (-1.95 0.89 2.41) (-2.05 -0.11 2.5) (-1.95 -1.11 2.41) (-1.65 -2.04 2.17) (-1.18 -2.84 1.77) (-0.57 -3.46 1.26) (0.15 -3.84 0.66) (0.91 -3.98 0.02) (1.53 3.35 0.75) (0.85 3.23 1.33) (0.21 2.89 1.87) (-0.34 2.34 2.33) (-0.77 1.62 2.68) (-1.03 0.78 2.91) (-1.12 -0.11 2.98) (-1.03 -1.01 2.91) (-0.77 -1.84 2.68) (-0.34 -2.56 2.33) (0.21 -3.11 1.87) (0.85 -3.46 1.33) (1.53 -3.58 0.75) (2.07 2.72 1.39) (1.5 2.62 1.86) (0.98 2.34 2.3) (0.53 1.89 2.68) (0.19 1.3 2.96) (-0.03 0.62 3.15) (-0.1 -0.11 3.21) (-0.03 -0.84 3.15) (0.19

-1.53 2.96) (0.53 -2.11 2.68) (0.98 -2.56 2.3) (1.5 -2.84 1.86) (2.07 -2.94 1.39) (2.47 1.89 1.88) (2.08 1.82 2.21) (1.71 1.62 2.52) (1.39 1.3 2.79) (1.15 0.89 2.99) (0.99 0.41 3.12) (0.94 -0.11 3.16) (0.99 -0.63 3.12) (1.15 -1.11 2.99) (1.39 -1.53 2.79) (1.71 -1.84 2.52) (2.08 -2.04 2.21) (2.47 -2.11 1.88))

:LINES ((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (8 9) (9 10) (10 11) (11 12) (13 14) (14 15) (15 16) (16 17) (17 18) (18 19) (19 20) (20 21) (21 22) (22 23) (23 24) (24 25) (26 27) (27 28) (28 29) (29 30) (30 31) (31 32) (32 33) (33 34) (34 35) (35 36) (36 37) (37 38) (39 40) (40 41) (41 42) (42 43) (43 44) (44 45) (45 46) (46 47) (47 48) (48 49) (49 50) (50 51) (52 53) (53 54) (54 55) (55 56) (56 57) (57 58) (58 59) (59 60) (60 61) (61 62) (62 63) (63 64) (0 13) (1 14) (2 15) (3 16) (4 17) (5 18) (6 19) (7 20) (8 21) (9 22) (10 23) (11 24) (12 25) (13 26) (14 27) (15 28) (16 29) (17 30) (18 31) (19 32) (20 33) (21 34) (22 35) (23 36) (24 37) (25 38) (26 39) (27 40) (28 41) (29 42) (30 43) (31 44) (32 45) (33 46) (34 47) (35 48) (36 49) (37 50) (38 51) (39 52) (40 53) (41 54) (42 55) (43 56) (44 57) (45 58) (46 59) (47 60) (48 61) (49 62) (50 63) (51 64)))

Appendix II.

The following is the representation of COMPARISON-OBJECT-A, discussed in Section 3.1 and illustrated in figure 5.

COMPARISON-OBJECT-A

(OBJECT

:POINTS ((-2.25 -1.27 0.0) (0.45 -1.27 -2.0) (0.72 1.81 4.0) (-1.98 1.81 -6.0) (-0.72 -1.81 8.0) (1.98 -1.81 -10.0) (2.25 1.27 12.0) (-0.45 1.27 -14.0))

:LINES ((0 1) (1 2) (2 3) (3 0) (4 5) (5 6) (6 7) (7 4) (0 4) (1 5) (2 6) (3 7)))

We can compare this object with its orthographic projection (left-hand image in figure 5). Unlike the other objects examined, this object is clearly *not* the object perceived when we look at the image. In fact, we know what object *is* perceived: it is CUBE-OBJECT, illustrated in figure 2. And we already know its complexity: 245. We would expect, therefore, that the complexity of COMPARISON-OBJECT-A will be greater than 245.

And this is indeed the case; the complexity of COMPARISON-OBJECT-A is 825, which would tend to explain why the vision system prefers CUBE-OBJECT over COMPARISON-OBJECT-A when looking at the image.

The image itself, as we know, has a complexity of 560, which would tend to explain why the vision system prefers CUBE-OBJECT to the image.